Load Distribution on a Heterogeneous Cluster of Workstations

Jose R. Santos Roman
Advisor:Dr. Jaime Seguel

Mathematics Department
University of Puerto Rico, Mayaguez Campus
Mayaguez, Puerto Rico 00681-5000
Email: jsantos@math.uprm.edu

Abstract

Clusters of workstations are becoming very popular because of their price performance advantages over super computers from Cray and IBM. Although programming clusters is not very complicated, some considerations should be taken when the execution is intended for a cluster of heterogeneous workstation. In fact, just dividing the amount work equally among different types of workstation might not yield optimal performance since different CPU process data faster than others. This work deals with design considerations for C code with MPI. The idea is developing code that distributes the workload not only by the number of processors in the cluster but also, the processing power of each CPU.

1. Introduction

The concept of load balancing is not reduce to simply distributing the amount of work among the processors on a heterogeneous cluster. In fact processors in a network cluster can have different speeds of execution. This paper presents some experiments on load distribution using a cluster with different architectures and speeds.

Let's say we build a two-node cluster. The first node is a Pentium 100 MHz and the second is a Pentium III Xeon 500 MHz. If we give each processor the same amount of work, the node with the Xeon processor will finish in a fraction of the time taken by the slower processor, and the Xeon processor will have to wait idle while the Pentium finishes. In this particular network cluster, the program might have executed faster on a single CPU than in both. This raises the question of scalability of a program.

The experiment conducted in this research tries to find methods for designing code able to distribute the workload on the basis of the execution speed of the processor rather than the amount processor in the cluster. By doing this we can easily build cluster that can grow over time with out architectural considerations. As the needs arise, we can just add a new node without worrying too much about its processor's speed or the architectures of the workstation.

2. Computer Environment

MPI stands for **M**essage **P**assing **I**nterface. This is a standard paradigm for both parallel machines with shared memory and distributed memory. MPI takes the best features of many message-passing implementations and creates a standard in which most computer vendors could agree on. This makes MPI programs more portable in terms of the

implementations. If one implementation doesn't work as expected, one could always try use another with minimal or zero code alterations. The standard allows C and FORTRAN 77 bindings for ease of use and convenience.

MPICH is a freely available implementation of the MPI standard that runs on a variety of systems. The experiments reported below were conducted on MPICH running under Linux.

Linux is a free Unix-type operating system originally created by Linus Torvalds with the assistance of developers around the world. Linux is an independent POSIX implementation and includes true multitasking, virtual memory, shared libraries, demand loading, proper memory management, TCP/IP networking, and other features consistent with Unix-type systems. Developed under the GNU General Public License, the source code for Linux is freely available.

3.The Cluster

The cluster used for this experiment consists of two nodes.

First node(Master):

Pentium II 233Mhz
32KB L1 Cache(CPU speed)
512KB L2 Cache(Half CPU speed)
66 MHz FSB
128 MB SDRAM 7ns
10base2 Network adapter
The rest of the hardware is not relevant for this experiment

Second node:

K6 200Mhz
64KB L1 Cache (CPU speed)
512KB L2 Cache@66MHz
66 MHz FSB
128 MB SDRAM 7ns
10base2 Network adapter
The rest of the hardware is not relevant for this experiment

Each node runs under Red Hat Linux 6.0 with latest updated patches and MPICH 1.1.2. The slave node reads the directories /home and /usr/local from the master node via NFS. User accounts are read from the master node via NIS.

4. Computation of Load Balancing

The first program used in the experiment calculates PI in **N** iterations. The program is actually use to determine the number of iterations that each processor could complete in an interval of 10 seconds.

```
startwtime = MPI_Wtime();
  h   = 1.0 / (double) n;
  sum = 0.0;
  while((endwtime-startwtime)<10){
     for( g=0; g<=10000; g++){
          x = h * ((double)i - 0.5);
          sum += f(x);}
          i++;
     endwtime = MPI_Wtime();
 }
  fprintf(stderr,"%d Number on Process %d
on %s\n",i,myid, processor_name);
```

Once completed, we divide the number of iterations by the lowest number to obtain a factor of performance. The slowest processor gets a performance factor of 1. This factor is later used on the program to achieve the load

balancing for each processor. Using the following function:

```
long int get_partition(int id,long int n,float
factor[],int numproc)
{
  float factor_sum=0;
  int i;
  long int x=0;
  for(i=0; I<=numproc; i++)
    factor_sum = factor_sum + factor[i];
  if(id<numproc){
    for (i=0; i<id; i++)
      x=x+factor[g]*(n/factor_sum);
    return x;}
  else return n;
}
```

we enter the id of the processor, number of iterations, the vector that contains the performance factor and total number of processors. The number returnd by this function is used in the program to decide the workload for each process that is executed in the cluster.

```
proc[0]=1.3971;
proc[1]=1.0;
h   = 1.0 / (double) n;
sum = 0.0;
for (i =
get_partition(myid,n,proc,numprocs)+1;
    i
<=get_partition(myid+1,n,proc,numprocs) ;
i++)
    {
     x = h * ((double)i - 0.5);
                    sum += f(x);
                    }
                    mypi = h * sum;
MPI_Reduce(&mypi, &pi, 1,
          MPI_DOUBLE, MPI_SUM, 0,
          MPI_COMM_WORLD);
```

The second program in this paper is simply a matrix by vector multiplication. This program generates a dynamic memory matrix on each processor. It uses the same factor number from the previous program, and studies the effects that different factors have on the programs execution time.

The main difference between this to program is that the matrix program is very memory intensive. The dimension of the matrix is 3000 rows and 3000 columns, the values of the matrix are type float. This gives a memory allocation of 72 MB of memory. In this problem, memory speed, FSB speed and Cache speed are very important factor to consider for the total time of execution.

5. Analysis of Results

The results are divided in several fields:

Procs: Number of processors in the cluster.
Proc_Fac #: Performance factor use for the processor #.
N: Number of iterations (Pi program only).
NxN: Matrix dimensions.
CPU Load: Processor load with respect to the execution time (Matrix program only).
Exec Time: Total time of execution.

Table 1. Results from Pi Program

Procs	Proc_fac 0	Proc_fac 1	N	Exec Time
1	1.0	N.A.	1000000	381 s
2	1.0	1.0	1000000	288 s
2	1.4	1.0	1000000	237 s

Table 2. Result from Matrix Vector Multiplication

Proc	Proc_fac 0	Proc_fac 1	NxN	CPU Load	Exec Time
1	1.0	N.A.	3000x3000	96 %	34 s
2	1.0	1.0	3000x3000	69 %	30 s
2	1.4	1.0	3000x3000	86 %	29 s
2	1.8	1.0	3000x3000	94 %	27 s

6. Conclusions

Viewing the results of Table 1 we can se that by using the factor obtained by the get_factor function, the time of execution of the Pi program is reduced considerably. Adding other numbers by hand resulted in larger execution times. Since the get_factor program uses the same arithmetic as the Pi program the performance was optimal. This wasn't the case of the Matrix program since the memory speed of the AMD system is slower than that of the Pentium II. The cache of the AMD system is also considerably slower since it working at 66 MHz versus 116.5 MHz. This made the execution time of the matrix program to decrease with the current factor value, but not to an optimal performance. Putting the factor by hand, we discovered that 1.8 had a better effect and produced an equal CPU load 94 percent on both nodes. The CPU load is directly proportional to the execution time, this means that with equal CPU loads each processor is working on full load on the same amount of time. This proves that the execution time is not related only to the power of the CPU and that other consideration such as memory, FSB, IO speed, network communications speeds and others, should be taken into consideration when performing load distribution in a cluster of workstations.

7. Future Work

Future plans for this project are to add a more advance factor system that takes the performance of different part of the cluster and stored them on a matrix instead of a vector. With this system one can optimize the code depending on the task that each algorithm will perform. The challenge consists of finding the right combination of factor to obtain optimal performance and thus reduce even further the execution time.

References

[1] "The MPI Forum"
http://www.mpi-forum.org/

[2] "MPI 1.1 Report"
http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html

[3] "An Introduction to MPI for Computational Mechanics"
P.K. Jimack and N. Touheed
Computational PDE Unit
School of Computers Studies
University of Leeds